Table of Contents

# Bab 5: Interfaces dan Type Aliases di TypeScript

## Penjelasan Materi

Interfaces dan Type Aliases adalah dua fitur penting di TypeScript yang memungkinkan kita untuk mendefinisikan struktur dan tipe data kustom. Interfaces berfungsi sebagai kontrak yang mendefinisikan struktur objek, sementara Type Aliases memungkinkan kita untuk membuat nama alternatif untuk tipe data yang kompleks. Kedua fitur ini sangat berguna untuk membuat kode yang lebih terorganisir, mudah dibaca, dan mudah di-maintain.

## Analogi yang Mudah Dipahami

Bayangkan Interface seperti blueprint kontrak kerja: - Interface adalah daftar persyaratan yang harus dipenuhi - Properties adalah kewajiban yang harus ada - Optional properties seperti bonus yang tidak wajib - Method signatures seperti tugas yang harus bisa dilakukan - Extending interface seperti menambah persyaratan tambahan

Bayangkan Type Alias seperti nama panggilan: - Type Alias memberi nama baru untuk tipe yang kompleks - Union types seperti pilihan menu paket - Intersection types seperti menggabungkan beberapa paket - Mapped types seperti mengubah semua item dalam menu

## Point Penting

1. **Interface Basics**
   * Property signatures
   * Method signatures
   * Optional properties
   * Readonly properties
2. **Interface Extension**
   * Extending interfaces
   * Multiple inheritance
   * Interface merging
   * Implementing interfaces in classes
3. **Type Aliases**
   * Basic type aliases
   * Union types
   * Intersection types
   * Literal types
4. **Advanced Types**
   * Mapped types
   * Conditional types
   * Utility types
   * Template literal types
5. **Best Practices**
   * Interface vs Type Alias
   * Naming conventions
   * Documentation
   * Reusability

## Contoh Kode dan Penjelasan

```typescript // 1. Basic Interface interface Pengguna { id: number; nama: string; email: string; umur?: number; // Optional property readonly createdAt: Date; // Readonly property }

// 2. Interface dengan Methods interface CRUD { create(data: any): void; read(id: number): any; update(id: number, data: any): void; delete(id: number): void; }

// 3. Interface Extension interface PenggunaAdmin extends Pengguna { role: “admin”; permissions: string[]; }

// 4. Interface Implementation class UserService implements CRUD { private users: Pengguna[] = [];

create(data: Pengguna): void {  
 this.users.push(data);  
}  
  
read(id: number): Pengguna | undefined {  
 return this.users.find(user => user.id === id);  
}  
  
update(id: number, data: Partial<Pengguna>): void {  
 const index = this.users.findIndex(user => user.id === id);  
 if (index !== -1) {  
 this.users[index] = { ...this.users[index], ...data };  
 }  
}  
  
delete(id: number): void {  
 const index = this.users.findIndex(user => user.id === id);  
 if (index !== -1) {  
 this.users.splice(index, 1);  
 }  
}

}

// 5. Type Aliases type ID = string | number; type Status = “active” | “inactive” | “pending”;

// 6. Complex Type Alias type ResponseData = { status: number; data: T; message: string; timestamp: Date; };

// 7. Union Types type Result = string | number | boolean;

// 8. Intersection Types type AdminUser = Pengguna & { role: “admin”; permissions: string[]; };

// 9. Mapped Types type Optional = { [P in keyof T]?: T[P]; };

// 10. Utility Types Usage type PartialPengguna = Partial; type ReadonlyPengguna = Readonly; type PickPengguna = Pick<Pengguna, “id” | “nama”>; type OmitPengguna = Omit<Pengguna, “email”>;

// 11. Advanced Usage interface ApiResponse { data: T; metadata: { page: number; total: number; limit: number; }; }

type ApiEndpoint = { get(): Promise<ApiResponse<T[]>>; getById(id: ID): Promise<ApiResponse>; create(data: Omit<T, “id”>): Promise<ApiResponse>; update(id: ID, data: Partial): Promise<ApiResponse>; delete(id: ID): Promise; };

// Implementasi class PenggunaAPI implements ApiEndpoint { async get(): Promise<ApiResponse<Pengguna[]>> { // Implementasi return { data: [], metadata: { page: 1, total: 0, limit: 10 } }; }

async getById(id: ID): Promise<ApiResponse<Pengguna>> {  
 // Implementasi  
 return {  
 data: {  
 id: 1,  
 nama: "John",  
 email: "john@example.com",  
 createdAt: new Date()  
 },  
 metadata: {  
 page: 1,  
 total: 1,  
 limit: 1  
 }  
 };  
}  
  
async create(data: Omit<Pengguna, "id">): Promise<ApiResponse<Pengguna>> {  
 // Implementasi  
 return {  
 data: {  
 id: 1,  
 ...data,  
 createdAt: new Date()  
 },  
 metadata: {  
 page: 1,  
 total: 1,  
 limit: 1  
 }  
 };  
}  
  
async update(id: ID, data: Partial<Pengguna>): Promise<ApiResponse<Pengguna>> {  
 // Implementasi  
 return {  
 data: {  
 id: Number(id),  
 nama: "Updated John",  
 email: "john@example.com",  
 createdAt: new Date(),  
 ...data  
 },  
 metadata: {  
 page: 1,  
 total: 1,  
 limit: 1  
 }  
 };  
}  
  
async delete(id: ID): Promise<void> {  
 // Implementasi  
}

}

// Penggunaan const api = new PenggunaAPI();

async function main() { // Get all users const users = await api.get(); console.log(users);

// Create new user  
const newUser = await api.create({  
 nama: "John Doe",  
 email: "john@example.com",  
 createdAt: new Date()  
});  
console.log(newUser);  
  
// Update user  
const updatedUser = await api.update(1, {  
 nama: "John Updated"  
});  
console.log(updatedUser);  
  
// Delete user  
await api.delete(1);

} ```

## Cara Kerja TypeScript Interfaces dan Types

1. **Type Checking**:
   * TypeScript memeriksa struktur objek
   * Memastikan implementasi sesuai interface
   * Validasi tipe saat compile time
2. **Type Inference**:
   * TypeScript dapat menebak tipe berdasarkan penggunaan
   * Bekerja dengan union dan intersection types
   * Membantu dalam type narrowing
3. **Code Generation**:
   * Interface dan types dihapus saat kompilasi
   * Tidak ada overhead di runtime
   * Hanya untuk development time checking

## Tips dan Trik

1. **Pilih Interface vs Type**

* // ✅ Gunakan interface untuk API public  
  interface UserAPI {  
   getUser(id: number): Promise<User>;  
  }  
    
  // ✅ Gunakan type untuk union/intersection  
  type Status = "success" | "error";

1. **Extend vs Intersection**

* // ✅ Interface extension untuk inheritance  
  interface Animal {  
   name: string;  
  }  
  interface Dog extends Animal {  
   bark(): void;  
  }  
    
  // ✅ Type intersection untuk composition  
  type Logger = Console & {  
   debug(message: string): void;  
  };

1. **Generic Constraints**

* // ✅ Membatasi tipe generic  
  interface HasId {  
   id: number;  
  }  
    
  function deleteItem<T extends HasId>(item: T): void {  
   console.log(\`Deleting item \${item.id}\`);  
  }

## Kesalahan yang Sering Dilakukan Pemula

1. **Mengabaikan Readonly**

* // ❌ Buruk: Property bisa diubah  
  interface Config {  
   apiKey: string;  
  }  
    
  // ✅ Baik: Property tidak bisa diubah  
  interface Config {  
   readonly apiKey: string;  
  }

1. **Terlalu Banyak Union Types**

* // ❌ Buruk: Union type yang kompleks  
  type Response = {  
   data: string | number | boolean | object;  
  };  
    
  // ✅ Baik: Gunakan generic  
  interface Response<T> {  
   data: T;  
  }

1. **Tidak Menggunakan Utility Types**

* // ❌ Buruk: Menduplikasi interface  
  interface UpdateUser {  
   name?: string;  
   email?: string;  
  }  
    
  // ✅ Baik: Menggunakan Partial  
  type UpdateUser = Partial<User>;

### Solusi:

1. Gunakan interface untuk API public dan kontrak
2. Gunakan type untuk union dan intersection
3. Manfaatkan utility types TypeScript
4. Terapkan readonly untuk immutable data
5. Dokumentasikan interface dan type dengan JSDoc